MTH6412B: Projet Voyageur de commerce
Phase 1

Éloise Edom (1555312)
Samuel Goyette (1983791)

19 septembre 2018

1 Justification de notre approche

1.1 Définition du type Edge

Nous avons défini les arêtes par leurs extrémités, soit deux noeuds, et leur poids car ce sont les informations pertinentes pour tracer un graphique approprié et pour exploiter ce graphe par la suite.

```
mutable struct Edge <: AbstractEdge
    node1 :: Node
    node2 :: Node
    weight :: Int
end
```

Ici nous n'avons pas défini de type pour Edge afin d'éviter les conflits avec le type de Node. Lorsque nous seront plus avancé dans le projet, nous pourront sélectionner le type approprié.

1.2 Ajout et affichage d'arête

Pour démontrer le fonctionnement des ces deux aspects, nous nous baserons sur un exemple de petite taille créé par nos soins. Lors de l'ajout d'une arêtes, il est nécessaire de spécifier les deux sommets qu'elle relie ainsi que son poids. L'ajout ce faire de la manière présentée ci-dessous. La fonction qui remplie ce rôle est similaire à celle pour les noeuds.

```
julia> arete1 = Edge(noeud1, noeud2, 5)
       Edge(Node{Array{Int64,1}}("1", [1, 2]), Node{Array{Int64,1}}("2", [4, 2]), 5)
```

Par la suite pour l'affichage des informations du graphe se fait de la manière suivante :

```
julia> show(G1)
       Node 1, data: [1, 2]
       Node 2, data: [4, 2]
       Edge connecting Nodes 1--2; Poid 5
       "Graph test1 has 2 nodes and 1 edges"
```

Cette structure d'affichage permet de résumer les informations essentielles, c'est-à-dire les noeuds et leurs coordonnées, les arêtes, leurs extrémités et leur poids.

1.3 Lecture des poids à partir des instances ".tsp" fournies et affichage de l'objet type graphe

Afin de prendre en compte le poids des arêtes, deux fonctions ont été modifiées : **read_edges** et **read_stsp**.

Plusieurs modifications ont été faites sur la première. D'abords, l'indice de la boucle présentée ci-dessous a été corrigé car il commençait à 0 au lieu de 1. Ensuite le poids est récupéré en même tant que les extrémités des arêtes. De plus, le `parse` est fait dans chacun des cas possible du `if` car la structure des matrices des instances fournies change, cela évite tous conflit.

```
 for j = start:start + n_on_this_line - 1
     n_edges = n_edges + 1
     if edge_weight_format in ["UPPER_ROW", "LOWER_COL"]
        edge = (k+1, i+k+1+1, parse(Int, data[j+1]))
     elseif edge_weight_format in ["UPPER_DIAG_ROW","LOWER_DIAG_COL"]
        edge = (k+1, i+k+1, parse(Int, data[j+1]))
 ...
 end # for
```

Nous avons testé notre programme avec l'instance "bayg29.tsp". Notre programme principal fonctionne de la façon suivante : on extrait les noeuds et arêtes brutes avec **read_stsp**, on tri les noeuds pour les mettre en ordre croissant, on mets les noeuds brutes dans la classe **Node** et on met les arêtes brutes dans la classe **Edge**. Finalement on mets les noeuds et arêtes dans un graph de la classe **Graph**. Lorsque nous appelons notre programme le résultat obtenu est :

```
julia> include("main.jl")
Reading of header :
Reading of nodes :
Reading of edges :
Node 1, data: [1150.0, 1760.0]
Node 2, data: [630.0, 1660.0]
...
Node 28, data: [1260.0, 1910.0]
Node 29, data: [360.0, 1980.0]
Edge connecting Nodes 1--2; Poid 97
Edge connecting Nodes 1--3; Poid 205
Edge connecting Nodes 1--4; Poid 139
...
Edge connecting Nodes 27--29; Poid 217
Edge connecting Nodes 28--29; Poid 162
"Graph [insert graph name here] has 29 nodes and 406 edges"
```

Ainsi, l'objet de type Graphe construit est complet.
Le codes est disponible à l'adresse : https://github.com/Goysa2/mth6412b-starter-code.